AIAA 2001-4235

RE-ENGINEERING LEGACY MISSION SCIENTIFIC SOFTWARE*

Charles D. Norton
Jet Propulsion Laboratory
California Institute of Technology
MS 168-522, 4800 Oak Grove Drive
Pasadena, CA 91109-8099 USA

Viktor K. Decyk
Department of Physics and Astronomy
University of California at Los Angeles
Los Angeles, CA 90095-1547 USA
and
Jet Propulsion Laboratory
California Institute of Technology

High Performance Computing Systems & Applications Group
Center for Space Mission Information and Software Systems

ABSTRACT

Many mission-critical scientific applications often rely on a legacy of software representing great intellectual and commercial value. This software is generally well debugged, produces trusted results, is actively meeting end-user goals, and preserves (sometimes hidden) expert knowledge that cannot be easily reproduced. Nevertheless, more ambitious missions require increased capabilities that impose new demands on software. Should these legacy codes be abandoned and rewritten from scratch, or can they be modernized to achieve new objectives?

Our approach for modernizing legacy scientific software, based on the new features of Fortran 90/95, will be presented. The methodology adds new capabilities, and increased safety, while promoting collaborative development and abstraction-based design. Application of this technique to modernize the Modeling and Analysis for Controlled Optical Systems software (developed at JPL and important to NASA's Next Generation Space Telescope Project) will be described concluding with new directions such as software tools for partial automation and evolution toward object-oriented concepts.

MODERNIZING SCIENTIFIC SOFTWARE

Legacy software has great value since it is generally well debugged, produces results that are trusted, and is actively meeting end-user goals. The amount of hidden expert knowledge embedded in such software can be significant making its preservation important.[6] Nevertheless, legacy software has limitations. It can be difficult to extend, modify, and it does not support collaborative development very well. This can impede the ability to meet new and expanded mission goals as timelines and budgets become tighter. One approach to this problem is to rewrite the software from scratch, but this may introduce more serious costs. In particular, developing new verification and validation tests can be expensive. Also, ensuring that the legacy code was faithfully rewritten, regardless of the programming language applied, cannot always be guaranteed.

Generally, if the functionality of the legacy software is sound, it can be wrapped in a modern interface where the original code is mostly unmodified. The idea of wrapping code means that the original legacy software is preserved while a new layer of software is introduced to separate the old software from the new software.[1] The wrapper provides the best means of retaining the functionality of the legacy software investment while providing a more flexible context from which new software, based on modern concepts, can be introduced. There are many benefits to this approach.

1. Software remains in productive use while applications are modernized.
2. Avoids costly and potentially harmful software rewrites.
3. Promotes collaborative development while resolving organization problems exhibited in older codes.
4. Re-engineering occurs more quickly than rewriting, while preserving verification and validation tests, especially

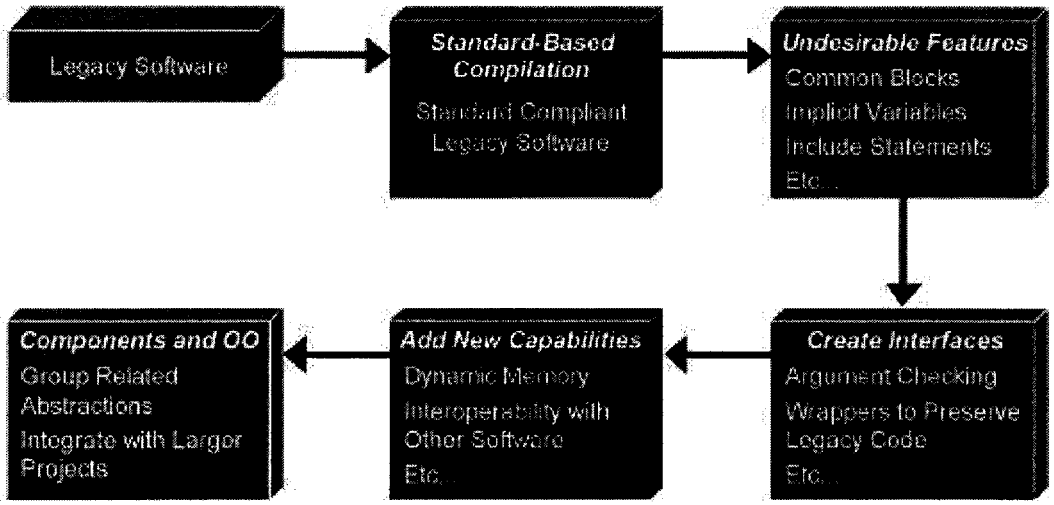

Figure 1. Legacy Software Modernization Process.

when the original programmers are involved.

5. Old bugs are uncovered that are often unknown accidents waiting to happen

Extending the functionality of legacy systems has become more important as modern applications increase in complexity and require the interaction of multiple contributors.

The Technology Applied

We have found that Fortran 90/95 has new features to support object-oriented principles beneficial for scientific programming and introduce a design methodology that defines a step-by-step process to modernize legacy application codes using state-of-the art software practices.[2,3] While we emphasize Fortran applications, due to the abundance of Fortran legacy codes, similar techniques can be applied to software written in other domain-specific or general-purpose languages, including C or C++.

Our process begins by upgrading the existing Fortran application to standard conforming Fortran 90/95. Next, interfaces to the original application routines are introduced to add safety features by detecting common programming errors. These interfaces ensure that the wrapper layer, added next, always correctly calls the legacy code. This wrapper layer allows problem based object abstractions to be introduced that interact cleanly with the legacy code, while supporting new enhancements. It also preserves the original, mostly unmodified, legacy software. The user can communicate with the modernized code across these layers and continuous development can occur simultaneously among these layers.

Many new features in the Fortran 90/95 standard provide benefits that are unfamiliar to experienced Fortran 77 software engineers. These features add safety, simplify complex operations, and allow software to be organized in a logically related way. Since backward compatibility is preserved one can incrementally make modifications while preserving existing work. Briefly, some of these new features are:

- **Modules**: Encapsulates (groups together) data, routines, and type declarations while providing accessibility across program units.
- **Use-Association**: Controls access to module content across program units.
- **Interfaces**: Verifies that the argument types in the procedure call match the types in the procedure declaration.
- **Derived Types**: User-defined types that support abstractions in programming. The creation of these types allows one to support problem domain based design.
- **Array Syntax**: This syntax simplifies whole array, and array subset, operations.
- **Dynamics**: Various kinds of dynamic structures are supported including allocatable arrays and pointers.

A very powerful realization is that combining these ideas allows support for object-oriented concepts. There are a number of textbooks on Fortran 90. One that we recommend is "Fortran 90 Programming", by Ellis.[4]

LEGACY MODERNIZATION PROCESS

The following process has been successfully applied for modernizing legacy software and it defines a plan of action for such projects. The diagram in figure 1 shows the fundamental stages involved. While we focus on Fortran legacy codes the same stages could be modified for software written in other languages. Many of the specific actions taken will also depend on the code structure and objectives.

Clearly Identify The Objectives

It is very important to have a conversation with the software owners to determine their objectives. The flowchart of the modernization process may help guide this discussion.

Understand The Legacy Software

Understanding, even at a basic level, how the legacy software is organized is valuable. While it is possible to perform the modernization without detailed knowledge of the application, knowing the design is very helpful. Here are some common questions that should be asked.

1. Is this a stand-alone application or is additional software required?
2. Is this a single language code or a multilanguage code?
3. What platforms are required?
4. Who is responsible for answering questions if legacy bugs are detected?
5. What kind of obsolete features exist in the software?
6. Are any third-party developers involved and is their software proprietary?

Addressing Undesirable Features

One of the most undesirable features of legacy Fortran 77 codes are COMMON blocks since they often inhibit more advanced features, like dynamic memory. They also discourage code sharing since everything is exposed. For this reason, modifying large common blocks can also be intimidating since inadvertent errors are easy to introduce. Other undesirable features include implicitly declared variables, which are dangerous, and include statements that are platform dependent based on how directories are specified.

Common blocks can be handled by placing the specification in a Fortran 90/95 module. Furthermore, rather than using include to make a textual substitution, the module information can become accessible using the Fortran 90/95 use statement in the appropriate routines.

The structure of the replacement is straightforward and is shown in figure 2. One could have simply copied the original common block from common.inc into a module exactly, but using the Fortran 90/95 constructs gives additional advantages. These include the ability to make the block members dynamic and the ability to add more functionality to the module by making other modules visible within its scope, to name a few.

Creating Interfaces

Interfaces are very important, as they add safety to the modernized software. They allow the compiler to verify consistent argument usage for procedures, which allows subtle errors to be detected and corrected in legacy codes.

```
C Original COMMON Block in common.inc
real arg1(10,10), arg2(10,10)
logical arg3
logical arg4
COMMON /BLOCK1/ arg1, arg2, arg3, arg4
SAVE /BLOCK1/

subroutine foo()
include 'common.inc'
...
end

! Modernized Version in common.f
MODULE common_block1
  implicit none
  save
  real, dimension(10,10) :: arg1, arg2
  logical :: arg3
  integer :: arg4
END MODULE common_block1

subroutine foo()
use common_block1
...
end subroutine foo
```

Figure 2. Example of converting a common block to a module.

Interfaces are created automatically for routines that are defined within modules, but we are currently interested in building interfaces for the legacy routines that will not be moved into modules at this time. Not every legacy routine requires an interface, but all of the routines accessible from the main program should have an interface. Furthermore, any routines in the scope of the main program that have arguments that will be dynamic will require an interface.

The interface statement is used to declare the procedure name and the types of its arguments. Since this is a Fortran 90/95 construct that will tie in the legacy code to the modernized code a new Fortran 90/95 interface.f file can be created to declare the Fortran 77 legacy interfaces. These interfaces can be placed in a module that in turn may use other modules, such as the common block modules recently created.

Figure 3 shows how the interface has exactly the same declaration as the original Fortran 77 legacy procedure; in fact it is best to just copy it explicitly. This means that when the legacy routine is called additional checks will be performed to ensure that the number and types of the arguments match exactly.

It may look like very little has been gained, but the benefit of the interface becomes clear when it is combined with a wrapper that allows more powerful Fortran 90/95 features to be applied.

```
! Interface Module in interface.f
MODULE interface_module
  USE common_block1
  implicit none
  save
  interface
    subroutine foof77(arg1, arg2, dim1)
    real arg1(dim1, dim1)
    integer :: dim1
    logical :: arg2
    end subroutine
  end interface
END MODULE interface_module
```

Figure 3. Specifying an interface to a legacy Fortran 77 routine.

For example, many Fortran 77 programs have very long argument lists because extra information must be included, such as the dimension of arrays. Since Fortran 90/95 arrays know their size these arguments do not need to be included in a wrapper function that calls the original legacy procedure, shown in figure 4.

```
! Interface Module in interface.f
MODULE interface_module
  USE common_block1
  implicit none
  save
  interface
    subroutine foof77(arg1, arg2, dim1)
    real arg1(dim1, dim1)
    integer :: dim1
    logical :: arg2
    end subroutine
  end interface
CONTAINS
  subroutine foof90(arg1, arg2)
  real, dimension(:,:) :: arg1
  logical :: arg2
    call foof77(arg1, size(arg1,1), arg2)
  end subroutine foof90
END MODULE interface_module
```

Figure 4. Creation of a wrapper to a legacy Fortran 77 routine where calls to the legacy procedure have been simplified.

This is a simple example, but the effect can be significant for very complex procedures. In fact, more functionality (such as dynamic memory) can be applied at this level using the wrapper while preserving the original legacy software. Furthermore, this can be achieved without a serious performance penalty when the legacy routine is non-trivial.

The interfaces can also clarify how Fortran 77 style arguments are sometimes passed to procedures. For example, it is not uncommon to find Fortran 77 programs that pass a two-dimensional array to a procedure that expects a one-dimensional array. This can cause compile errors when interfaces are used because they require that the arguments must match exactly. In such instances it is possible to create multiple interfaces to recognize this difference using a generic procedure to allow a single name to select the correct module procedure based on the argument list.

Adding New Capabilities

Now that the interfaces have been created and wrappers have been introduced to encapsulate the legacy software new capabilities can be added. For most legacy software the most desirable feature is dynamic memory. Fortran 90/95 supports many kinds of allocatable structures and they are straightforward to use. Dynamic memory increases the flexibility of the software since this frees the application user from fixed problem sizes. Interoperability with more modern software can also be achieved since the wrappers can be designed to utilize such applications. These new capabilities can be added without affecting the use of existing systems.

```
! Legacy Fortran 77 include of static COMMON data
parameter (mdttl=128)
integer nElt, RayID(mdttl,mdttl), ...
COMMON /EltInt/ nElt, RayID, ...
SAVE /EltInt/

! New Module for COMMON data
MODULE elt_common
  implicit none
  save
  integer :: nElt, mdttl = 128
  integer, allocatable, dimension(:,:) :: RayID
CONTAINS
! Constructor
  subroutine new_elt_common()
    allocate( RayID(mdttl,mdttl) )
  end subroutine new_elt_common
END MODULE elt_common
! Dynamic allocation from main program
PROGRAM example
  USE elt_common
  implicit none
  call new_elt_common
  ...
END PROGRAM example
```

Figure 5. Specifying an wrapper to a legacy Fortran 77 routine.

The example in figure 5 shows a legacy Fortran 77 common block with static data can be reorganized to support dynamic memory. This occurs by moving the common block into a module and specifying which structures will be dynamic. A constructor can be created to perform the allocation of the dynamic structure and this constructor can be called from the main program. A number of additional safety features such as checking if the structure was already allocated, handling of exceptional conditions like insufficient memory, and so forth can be added as well.

Toward Components And Object-Oriented Design

Fortran 90/95 contains derived types, like structures in C, which allow users to create their own types. This allows one to program using designs that better represent the problem domain. One of the major benefits of the methodology is that one can incrementally evolve the legacy code toward such a design while preserving the functionality of the legacy software. An object-oriented design allows the implementation details to change without affecting the user. In a sense, the interfaces and wrappers have hidden the details of the legacy software, but we can enhance the wrappers to support derived types evolving the code toward an object-oriented, component-based, design. Considering object-oriented issues has received libited attention in the past, but large applications have not been emphasized.[3,7]

```
! Creating derived types for object-based design
type species
  real, dimension(:,:), pointer :: coords
  real, charge_to_mass, kinetic_energy
end type species

! Using a legacy routine through an OO wrapper
subroutine w_push(particles, force, dt)
type (species) :: particle
real :: dt, qbm, wke
integer :: ndim, nparticle, nx
  ndim = size(particle%coordinates, 2)
  nx = size(force)
  qbm = particle%charge_to_mass
  wke = particle%kinetic_energy
  call push(particle%coords, force, qbm, wke, ndim
    nparticle, nx, dt)
end subroutine w_push
```

Figure 6. An object-based wrapper to a legacy Fortran 77 routine.

The example in figure 6 shows a legacy `push(...)` routine, wrapped by a Fortran 90/95 `w_push(...)`, routine that uses a derived type to group together related information. This was not possible in Fortran 77 so long complicated argument lists were required. The species type has a dynamic component, and other information, which simplifies the programmer's view of the data. Nevertheless, the original legacy software can still provide the functionality required.

In fact, a class can be created which groups together operations common to the new species type where the class member routines utilize legacy software internally. New software can be added to the class as well. This is a very powerful concept, but careful planning is always required when building an object-oriented design. This is illustrated in Figure 7.

```
! Creating classes for object-based design
MODULE plasma_class
! Create Derived Types...
CONTAINS
  subroutine new_species(...) ! Constructor...
  subroutine w_push(...) ! Class Members...
END MODULE plasma_class
```

Figure 7. Creation of a class framework containing wrappers and new routines.

Once the classes have been designed, and tested, the modules can be incorporated into the main program and calls to the member routines can replace calls to the original legacy software. Since interfaces for the legacy software still exist this process can be incremental, the software still works at the end of the day, and development can continue during the modernization process allowing existing objectives to be satisfied.

MACOS CASE STUDY

The Modeling and Analysis for Controlled Optical Systems (MACOS) software is an important NASA code that has been used for numerous projects. This software, developed by Dr. David Redding and others from the Optical Systems Modeling Group at JPL, provides powerful optical analysis tools and a unique capability for system-level design and analysis tasks. MACOS has many features, but a short list includes:

- Modeling optics on dynamic structures, deformable optics, and controlled optics
- Efficient general ray-trace capabilities
- Integrated support with other tools to create an end-to-end instrument system model

MACOS is written primarily in Fortran 77 and it interoperates with Matlab, PGPlot, and FFTw. There is also a subroutine library called SMACOS based on MACOS. Previous efforts to rewrite the software completely in C++ (to meet new objectives) were abandoned primarily because the new code did not perform as desired, and the designers are more fluent in Fortran.

The objectives of the designers were to achieve Fortran 90/95 standard compliance, dynamic memory support, and